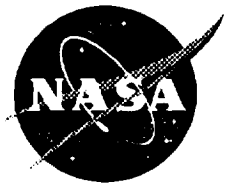
NASA Contractor Report 194943

ICASE Report No. 94-54



ICASE

DIRECTIONS IN PARALLEL PROGRAMMING: HPF, SHARED VIRTUAL MEMORY AND OBJECT PARALLELISM IN pC++

Francois Bodin
Thierry Priol
Piyush Mehrotra
Dennis Gannon

Contract NAS1-19480
June 1994

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23681-0001

Operated by Universities Space Research Association

Directions in Parallel Programming: HPF, Shared Virtual Memory and Object Parallelism in pC++*

François Bodin[a] *Thierry Priol*[a] *Piyush Mehrotra*[b] *Dennis Gannon*[c]

[a]Irisa, University of Rennes, France
[b]ICASE, MS 132C, NASA Langley Research Center, Hampton VA. 23681 USA
[c]Department of Computer Science & CICA Indiana University, Bloomington, Indiana, U.S.A.

Abstract

Fortran and C++ are the dominant programming languages used in scientific computation. Consequently, extensions to these languages are the most popular for programming massively parallel computers. We discuss two such approaches to parallel Fortran and one approach to C++. The High Performance Fortran Forum has designed HPF with the intent of supporting data parallelism on Fortran 90 applications. HPF works by asking the user to help the compiler distribute and align the data structures with the distributed memory modules in the system. Fortran-S takes a different approach in which the data distribution is managed by the operating system and the user provides annotations to indicate parallel control regions. In the case of C++, we look at pC++ which is based on a concurrent aggregate parallel model.

*This research is supported by DARPA under contract AF 30602-92-C-0135 from Rome Labs, National Science Foundation Office of Advanced Scientific Computing under grant ASC-9111616 and Esprit BRA APPARC and by the National Aeronautics and Space Administration under NASA contract NAS1-19480 while one of the authors was in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681.

1 Introduction

Exploiting the full potential of parallel architectures requires a cooperative effort between the user and the language system. There is a clear trade-off between the amount of information the user has to provide and the amount of effort the compiler has to expend to generate optimal parallel code. At one end of the spectrum are low-level languages where the user has full control and has to provide all the details while the compiler effort is minimal. At the other end of the spectrum is sequential languages where the compiler has the full responsibility for extracting the parallelism. Clearly, there are advantages and disadvantages to both approaches.

Explicit-tasking Languages

Current programming environments for parallel machines follow the first approach providing low-level constructs such as message-passing primitives as their principal language constructs. In such programming environments, an algorithm is specified as a set of sequential processes which execute concurrently, synchronizing and sharing data explicitly via messages. Since such environments directly reflect the underlying hardware, such an explicit-tasking approach allows the user to effectively exploit the full potential of the machine.

However, for data parallel algorithms, as one typically finds in scientific programming, such environments have proven quite awkward to use. The basic issue is that programmers tend to think in terms of synchronous manipulation of distributed data structures, such as grids, matrices, and so forth, while the languages available provide no corresponding language constructs. The hardware support for most current architectures is such that locality of data is critical for good performance. Thus, the programmer must decompose each data structure into a collection of pieces, with each piece "owned" by a single processor. All interactions between different parts of the data structure must then be explicitly specified using the low-level data sharing constructs such as message-passing statements supported by the language.

Decomposing all data structures in this way, and specifying communication explicitly, leads to programs which can be extraordinarily complicated. Experience has shown that message-passing versions of algorithms can be five to ten times longer than the sequential version. This code expansion hides the original algorithm among the details of low-level communications. Programs written in low-level languages also tend to be highly inflexible, since the partitioning of the data structures across the processors must be incorporated in all parts of the program. Each operation on a distributed data structure turns into a sequence of "send" and "receive" operations intricately embedded in the code. This "hard wires" all algorithm choices, inhibiting exploration of alternatives, as well as making the parallel program difficult to design and debug.

Direct Compilation of Conventional Languages

The second approach to programming multiprocessors, direct compilation of conventional languages for parallel execution, provides a number of important advantages. First, it allows programmers to continue using familiar languages as they move to newer and more complex machines. Second, there is a large body of existing programs which can be transported to parallel architectures without change. Third, the details of the target architecture are invisible to the programmer, so the complex

load-balancing and program design issues, which must be faced with the explicit-tasking languages, are not present.

This approach is, in a real sense, a direct outgrowth of successful research in construction of vectorizing compilers. and is currently being actively explored by several research groups [6, 7, 26, 51, 56, 63, 65]. Since the millions of lines of existing sequential programs cannot be easily replaced, nor are they readily modifiable, there is clear importance to this approach, and it will surely continue.

There are, however, a number of difficulties with this approach. The major one is that the semantics of conventional languages strongly reflects the sequential von Neumann architecture, making the task of automatic restructuring very difficult. Extracting parallelism from such programs requires very aggressive data-flow analysis including array subsection and inter-procedural analysis. Moreover, existing languages, especially Fortran, encourage programming styles which make it extremely difficult for compilers to extract much parallelism. Freely "equivalenced" arrays, and passing of "pointers" to simulate dynamic allocation, severely limit the compiler's ability to extract parallelism.

Also, once the parallelism has been exposed, it has to be mapped onto the target architecture. The appropriate mapping, including the distribution of data and work across the processors, is critically dependent on the characteristics of the program and also that of the target machine. Because the general mapping problem has been shown to be NP-complete the heuristic algorithms used tend to generate sub-optimal code. Given all these problems, the end result seems to be that direct compilation of sequential languages can extract only modest amounts of loop-level parallelism.

The Alternative: Modest Language Extensions

As argued above, the state of the art in advanced compiler design is not yet up to the task of parallelizing a sequential application for execution on a massively parallel system with a complex memory hierarchy. Consequently, the programmer must participate in this process. While there is a wide variety of new parallel programming languages that help solve this problem, we will focus attention on three approaches based on modest extensions and annotation systems for Fortran and C++. The goal of each system is to provide high performance and portability across the three prevailing classes of computer architectures which are distinguished by the memory model they present to the programmer.

True Shared Memory. The address space is global to the machine and access to memory is uniform. Examples of this class include the CRAY C90 and the SGI Power Challenge Series.

Shared Memory with Non-Uniform Memory Access (NUMA). The address space is global to the machine and access to memory is non uniform, i.e., access time depends on the address and the processor doing the data access. Examples of this class include the BBN TC2000, the CRAY T3D and the Convex MPP.

Distributed Memory Architecture: The address space is local to each processor and access to remote data is done usually via message passing. Examples here include the Intel Paragon, nCube, Parsytec, Meiko and Thinking Machines CM-5.

Because the first two categories provide a global address space for referencing data, they are the closest to the model most familiar to users. Consequently, the most direct way to make all three share this property is to build an operating system layer for the distributed memory machines that provides a "shared virtual memory" model on top of the native message passing system. Given such a system, any Fortran or C program can execute without modification on the machine. The problem is then reduced to providing a way for the compiler to partition parallel loops and schedule access to shared objects. Fortran-S, designed at IRISA is one such system. In the paragraphs that follow, we shall describe many of the important ideas that go into the construction of a shared virtual memory operating system and the Fortran-S compiler.

Fortran-S uses program annotations to partition control and the operating system automatically partitions the data. An alternative strategy is to ask the user to specify the way the data should be partitioned and have the compiler decide how to partition the control. High Performance Fortran (HPF) follows this approach. While HPF code could be compiled for a shared virtual memory, most systems will use the compiler to generate explicit message passing on distributed memory machines. In the third section of this paper we describe the HPF model and the language annotations and extensions required to implement it.

A third approach which combines the features of both HPF and shared virtual memory is pC++ which is based on a language extension, called concurrent aggregates, which allows the programmer to define a set of distributed objects which may be referenced from any processor in this system. As with HPF, the user provides information to the system about how the data objects should be partitioned among the system memory modules. However, communication between objects uses a mechanism based on the SVM paging model, but instead of migrating pages of data, copies of data objects are migrated. In the last section of this paper we describe pC++ and its execution model.

In this paper we have not described other promising approaches. Among these are functional programming languages such as SISAL [43] and ID, and task parallel systems such as CC++ [31], Linda [5], Fortran-M or that proposed in [13].

2 Fortran-S and Shared Virtual Memory

Programming with shared-memory or NUMA is usually simpler than programming distributed memory architectures because they offer a global view of the memory where distributed memory architectures let the user deals with data exchanges between processors by means of messages passing. Shared memory architectures are attractive from the programming point of view but they cannot afford scalability. As the number of processors increases the cost of the switch used to connect memory to processors increases very fast, and may even not be built at the required speed. Fully distributed memory architectures, on the other end, are scalable but do not offer to the programmer a single address space, making programming more complex.

For programming distributed memory architectures, approach as HPF proposes a global address space to the programmer and fills the gap between the programmer model and the machine by using sophisticated compiler techniques and the help from the programmer who is in charge of specifying at a high level the data distribution on the processors. Another alternative consists in providing the functionalities of a shared memory (it becomes virtual in this case) either implemented using hardware support or using operating system support. This approach makes distributed memory

architectures look like NUMA architectures which makes programming simpler and the compiler much easier to design.

2.1 Shared Virtual Memory

A Shared Virtual Memory (SVM) provides to the user an abstraction from an underlying memory architecture of a distributed memory parallel computer (DMPC). This memory abstraction is also named VSM (Virtual Shared Memory), DSM (Distributed Shared Memory) [37], etc. We will use SVM to name this memory abstraction. An SVM [39] is somewhat similar to the one which is currently used on classic mainframe computers. However, it differs in the fact that this virtual memory is shared by several processors. It provides a virtual address space that is shared by a number of processes running on different processors of a distributed memory parallel computer. The virtual address space is made up of pages* which are spread among local processor memories according to a mapping function. Compared to a global address space available on shared memory parallel computers (SMPCs), SVM relies on page caching and heavily on spatial locality. A global address space, like the one available on the BBN, usually allows access to a single word through the use of a fast interconnection network. In most cases DMPCs are loosely coupled architectures that have a high latency network. Accessing the data at a page level absorbs this high latency when spatial locality is exposed. Since the granularity of the data accesses is a page, several problems arise. For example, how does one keep pages coherent that are stored in several caches? How do we locate an up-to-date copy of a given page within the architecture? What happens when there is not enough room in a cache?

The first problem is related to *cache coherence*. Since processors may have to read from or to write to the same page, several processors have a copy of a page in their cache. If one processor modifies its copy, other processors run the risk of reading an old copy. A cache coherence protocol is needed to ensure that the shared address space is kept coherent [12]. A memory is considered to be *strongly coherent* if the value returned by a read from a location of the shared address space is the value of the latest store to that location [12]. In most cases, implementation of strong coherence in a SVM for DMPCs is based on an *invalidation* mechanism. It assumes that there is only one copy of a page with write access mode at a given time or, if there is multiple copies of a page, each of them are in read-only access mode. The processor that has written most recently into the page is called the *owner* of the page. When a processor needs to write to a page, that is not present in its cache or is present in read-only mode, it sends a message to the owner of the page in order to move it to the requesting processor. It then invalidates all the copies in the system by sending a message to the relevant processors. This *invalidation* strategy seems to be the best approach for DMPCs. The faulting management mechanism of a MMU is sufficient to implement this approach efficiently.

The second problem is called *page ownership*. When a processor needs to access a page, either in write or read access mode, which is not located in its cache, it must ask the owner to send it a copy of the page. This problem is related to the cache coherence protocol described previously. With the invalidation protocol, there is always one owner for a page and the ownership changes according to the page requests coming from other processors. Therefore, the problem is how to locate the current owner of a given page considering that the owner of a page changes. A solution is to update

*the granularity afforded by hardware virtual memory

a database that keeps track of the movement of pages in the system. This database can be either *centralized* or *distributed* [39]. In the centralized approach, a processor (called the manager) is in charge of updating the database for every page. When a processor needs a page, it sends a request to the manager which forwards the request to the owner of the page. Consequently, the manager is aware of all the movement of pages in the system. However, it may be a bottleneck since the manager processor receives requests from all other processors. The user's process running on the manager will be interrupted frequently and this approach will also create potential contention in the network. Distributing the database over several processors is a means to avoid these drawbacks.

The last problem is called page swapping. The problem arises when a processor is the owner of all the pages located in its cache and there is no more space in the cache. If it requests a new page, it has to find space in its cache. It cannot throw away a page from its cache since it owns all the pages. Therefore, pages have to be moved on an external high speed storage device, like disks. Some implementations, such as KOAN [34], use the other local memory for swapping page. However, the size of the virtual address space is bounded by the sum of all local memory.

The implementation of SVM mechanisms is done mostly by software: page requests are processed by the operating system running on each node. This implementation involves a substantial overhead since, user's processes have to be stopped by the operating system to resolve page requests. This task could be done by using dedicated VLSI hardware such is done with the KSR [2] machine and SCI bus-based parallel architectures [1] such as the new Convex MPP

2.2 Why Shared Virtual Memory May Not Work

Shared Virtual Memory has many intrinsic problems. In the following paragraphs, we discuss some of them.

Initial Page Distribution

The initial page distribution may lead to cold start misses, however this has a marginal effect on performance. After the beginning of the application, pages migrate to processors according to data accesses.

Page Thrashing

Page thrashing can lead to capacity misses. For instance consider the following loop:

```
doall i =1,n
  do j = 1,n
     A(j,i) = f(..., B(i,j),...)
  enddo
enddo
```

Due to the Fortran column-wise data layout, each access to matrix B will create a page fault if n is large enough. In that case interchanging the loop would not help, but loop blocking would.

False Sharing

False Sharing occurs when more than one processor, at a time, writes to the same page. The strong coherence mechanism ensures that each processor writing into a page sees the last modification of it. For example, consider the following loop.

```
doall i = 1,n
  A(i) =  f(.....)
enddo
```

Assuming that A is allocated in a shared address space and is only stored into one page, when increasing the number of processors the page will exhibit a **ping-pong phenomena**. That is, the page will move back and forth between processors, each write costing one page fault at worst (each word written will cost a data transfer of the size of the page). The execution of the loop becomes sequential because a page manager will serve only one page request at a time. This phenomena may severely degrade performance. However by increasing the size of the vector this phenomena may become negligible.

Barrier

When programming using message passing, synchronization between processes comes for free; data exchanges synchronize processes. When using shared variables, synchronization must be inserted to ensure data dependences between processes. However synchronization does not have to be implemented using shared variables. Most system support some sort of barrier operation which can be used as the primary synchronization mechanism. If the barrier is too slow, serious performance problems may result.

Broadcast

A drawback of shared virtual memory on DMPCs is its inability to run efficiently parallel algorithms that contain a producer/consumers scheme. In these cases, a page is modified by a processor and then it is accessed by the other processors. Since all page requests are sequentially processed by a page manager the accesses to the data are done sequentially. This obviously constitutes a serious bottleneck when the number of processors grows.

2.3 Why Shared Virtual Memory May Work

Shared virtual memory may work surprisingly well (see section 3.3) for the following reasons.

Vectorized Page access to data (block transfer)

Transferring a page makes efficient use of the network, masking most network latencies. There is clearly a tradeoff when choosing the page size. A large page size makes efficient use of the network, but increases the amount of unnecessary data transferred and false sharing becomes a greater problem. A small page size transfers a greater percent of useful data and decreases false sharing, but it makes inefficient usage of the network. To deal with a small transfer size on the KSR, where subpages of 128 bytes are the basic transfer unit, prefetch and poststore facilities are used to hide large access latencies.

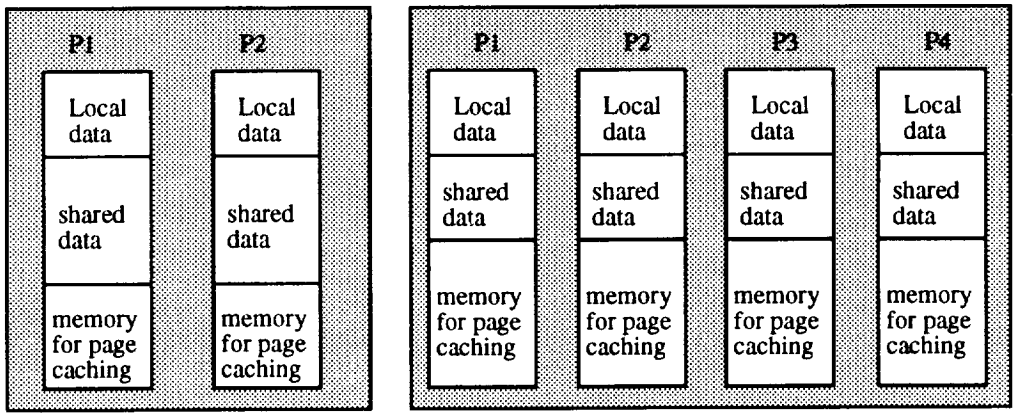


Figure 1: Allocation of data in memory, assuming 2 processors and assuming 4 processors, in the case of 4 processors more memory can be devoted to page caching

Page caching allows the exploitation of data locality

Page caching is the only way to compensate for the cost of moving a page between processors. This decreases the size of the effective SVM. Indeed the page caching memory is the remainder of the memory not used by the local data and the shared data. At some point if data are too big the remaining memory may be too small to keep pages necessary for an algorithm to behave efficiently. The main advantage of SVM over other mechanisms is that, even when locality properties of the program cannot be discover at compile time, the SVM can still exploit it. In this respect Shared Virtual Memory addresses the same problem as the Parti inspector/executor scheme [66].

All variables do not have to be shared

Only variables that are subject to parallel computation should be shared. Sharing all variables leads to very inefficient code.

Not all parallel computations depend on the SVM

For example, exploiting reduction parallelism is usually not done using shared virtual memory. Instead it can be implemented by the compiler by using message passing.

The Compiler Can Help A Lot

Compiler technique can help by optimizing programs so that they make better use of the virtual shared memory and also by decreasing the number of synchronizations in the program (i.e. decreasing the number of barriers).

2.4 Parallel Loop Scheduling and Shared Virtual Memory

Parallel loops scheduling is a critical issue in a programming environment that relies on shared virtual memory. Data movements are in charge of the system/hardware where loop scheduling is in charge of the compiler/user. Good scheduling has to ensure data locality and load balancing. Bad loop scheduling may result in many unnecessary page migrations, false sharing or an unbalanced

load. It should be noted that techniques such as self guided scheduling are not very well suited to shared virtual memory, because as the execution of a parallel loop proceeds, the size of blocks of iterations that are assigned to processors decreases. Consequently, false sharing increases. However, when associated with a cache coherence protocol that allows concurrent access to the same page, this technique may become adequate if it can be implemented efficiently on massively parallel distributed memory architecture.

There are two main scheduling techniques that are well suited to shared virtual memory because they can be used to reduce data movements. The first technique addresses the problem of false sharing especially when strong coherence protocols are used, and the second technique is concerned with data reuse across loops. In addition, they can be used together to provide good locality and to decrease the false-sharing:

Page Aligned Scheduling

Page aligned scheduling can be used to reduce false sharing. The principle consists of distributing iteration such that chunks of iterations allocated on processors are aligned with page boundaries. For example, if we use a simple block scheduling strategy of a simple loop:

```
do i = 1, N
   A[i] = A[i] + ...
enddo
```

we get:

```
bf = ceiling(N/P)
doall pid = 1,N,bf
   do i= pid,min(pid+bf-1,N)
     A[i] = A[i] + ...
   enddo
enddo
```

If N/P is not a multiple of the page size there will be false-sharing for each page shared by two processors. A simple solution in that case is to consider a blocking factor bf that take into account the page size (assuming $A[1]$ is aligned on a page boundary):

```
bf = ceiling(ceiling(N/pagesize)/P) * pagesize
doall pid = 1,N,bf
   do i= pid,min(pid+bf-1,N)
     A[i] = A[i] + ...
   enddo
enddo
```

However this technique does not always balance the workload. In general, this technique works well when the amount of data is large. A more complete description of the method is given in [25].

Affinity Scheduling

The affinity scheduling tries to minimize data movement by allocating iterations to processors according to data location [42]. An affinity scheduling technique is provided on the KSR1 machine.

It should be noted that dynamic scheduling, to improve load balancing, can be implemented with SVM but on distributed memory architectures the implementation of such a technique is usually very costly at runtime.

2.5 Compiler Optimizations for SVM

Compiler optimization for Shared Virtual Memory consists in increasing data locality, and thus minimizing data transfers. Optimization consists of removing shared variables as much as possible, changing the data layout, and applying loop transformations to improve locality without killing the load balance.

Removing Shared Variables

In some cases, it is possible to localize shared variables. The idea behind this optimization relies on the compilers capability of detecting access to data structure that are disjoint between processors. Compiler techniques used in this case are very close to the ones used for compiling FortranD and HPF programs. [62, 30].

Array Padding and Data layout

The Array Padding operation consists of extending array dimensions such that dimensions of the array are aligned with page boundaries. This reduces false-sharing because different vectors of the array do not share any pages. The main disadvantages of this technique are that it wastes memory (and so decreases the size of the memory that can be allocated to the cache) and also that this may increase the amount of communication (unused data are loaded when accessing useful data). More generally, data layout optimization tries to store data so that it minimizes false sharing [61, 18].

Optimizing Data Locality

Optimizing data locality relies on changing the access order to data structure so that it increases the spatial locality of a loop or it exploits better temporal locality. Loop transformations like loop interchanging, blocking, unimodular transformation may be used. When temporal locality exists, it may be possible to exploit data locality using localization of a portion of an array section that is subject to reuse. These techniques are common to global address space optimization and cache locality optimization. For example, considering the following loop.

```
doall i=1,n
  do j = 1,n
   do k =1,n
     A(k,i) = f(...,A(k,i),...)
   enddo
  enddo
enddo
```

It can be transformed into

```
doall i=1,n
  do t=1,n
    temp(t) = A(t,i)
  enddo
  do j = 1,n
   do k =1,n
     temp(k) = f(...,temp(k),...)
   enddo
  enddo
  do t=1,n
    A(t,i) = temp(t)
  enddo
enddo
```

where *temp* is allocated locally on processors. This optimization may reduce the number of page faults and the false-sharing. It should be noted that the array *temp* is the reference window as defined in [21, 9]. The cost of the copy is amortized by exploiting the temporal locality. However if there was no page thrashing and no false-sharing on array A in the original loop, there is no gain in using this transformation. When applying this kind of optimization, the size of temporaries must be limited. These techniques [63, 4, 41, 3, 23, 64, 55, 16, 44] are well known but usually targeted for hardware cache or local memory. Most of these techniques should be revisited to take into account the characteristics of shared virtual memory and in particular the false sharing phenomena.

Barrier Removal

When programming with a shared memory model (especially when the execution model is SPMD) synchronization between processes relies on barriers. One optimization the compiler can perform is to decrease the number of synchronization in the program. More generally, part of the optimization process consists in removing, as much as possible, calls to the runtime system.

2.6 Runtime Optimization for SVM

In some case, support for optimizations may come from system capabilities:

Weak Coherence

Several weak cache coherence protocols have been studied in the past. Each of them has some properties that can be exploited in a specific context [50, 57]. A modified version of the strong coherence protocol can be considered as a weak cache coherence protocol. If data accesses are made in different memory locations, it allows processors to modify their own copy of a page, without invalidating copies in other processors. When restoring the strong coherence protocol, all the copies of a page which have been modified are merged into a single page that reflects all the changes. From the programmer's point of view, the memory is always strongly coherent at a word level but is weak coherent at a page level. However such weak coherence scheme does not come

for free; its cost depends (usually linearly) on the maximum number of page copies there are to merge at end of the page weak coherence phase there is to perform. Weak Coherence protocol can be used for parallel loops because there is no data dependence between iterations of the loops and so no several writes to the same word of a page are performed by several different iterations.

Page Broadcast

Producer/consumers scheme can be efficiently managed by using the broadcasting facility of the underlying topology of DMPCs (hypercube, 2D-mesh, etc.). All pages that have been modified by the processor in charge of running the producer phase are broadcast to all other processors that will run the consumer phase in parallel. Since the producer has to keep track of all pages that have been modified, two new operating system calls have to be added in the user's code in order to specify both the beginning and the ending of the producer phase.

Page locking

Page locking allows a processor to lock a page into its cache until it decides to release it. This basic mechanism can be used to implement atomic update in a memory location. The user is responsible for adding two system calls that specify the beginning and the ending of the code section where each remote data access requires a page to be locked into the cache. Page locking is very efficient and minimizes the number of critical sections within a parallel code. On loosely coupled parallel architectures, such as DMPCs, using critical sections are time expensive. To illustrate this, let us take a small example such as a matrix assembly found in finite element applications. A loop is used to scan an irregular mesh and values are accumulated into a matrix. Access to this matrix is made by through an index scheme and there are often runtime data dependences. Consequently the loop can be parallelized if the accumulation is executed within a critical section to avoid multiple processors writing at the same time to the same matrix element. A page locking mechanism can replace many critical sections. Before updating a matrix element, the page that contains the matrix element is locked into the cache and then release after the update. The cost of such synchronization is simply related to the number of processors that access to the same page at the same time.

2.7 Mixing Messages and Shared Virtual Memory

Mixing of message passing and shared variables can be used to improve performance in library code. When dealing with shared variables and messages, programming is somewhat simplified since the programmer does not have to worry about the data distribution. The programmer only has to think in term of parallel processes. One of the main advantage of this approach, is that an efficient algorithm may be implemented independently of the program it is called from. For example, consider the in-place matrix transpose. This algorithm behaves very badly with SVM when data transfer is at the level of pages. But by using message passing to do the transpose, it is possible to get speedup on this operation. The algorithm can be written so it is independent of the data distribution of the matrix. In a pure message passing programming environment, it is not possible to provide such a primitive without forcing the programmer to use a predefined data distribution of the matrix on the processor, this data layout that may be completely inadequate in the remainder of the application.

3 Fortran-S: a Prototype Environment for SVM

Fortran-S is a Fortran programming environment that relies on the shared virtual memory KOAN. The programming model is based on shared variables and parallel loops. Parallel loops and shared variables are declared to the compiler via directives. The project's main goal is to study compiler and programming environment for shared virtual memory. Fortran-S differs from the KSR-Fortran mainly in the execution model. KSR-Fortran relies on fork-and-join execution (i.e. the main thread is spawn in multiple threads when parallel phases of execution occur) where Fortran-S relies on a SPMD execution model (i.e. a thread is created on every processor during the loading phase). To illustrate Fortran-S, we provide the following small example:

```
      real v(n,n)
C$ann[Shared(v)]
      do i = 1,n
         tmp = 0.0
         do k = 1,n
            tmp = tmp + v(k,i)*v(k,i)
         enddo
         xnorm = 1.0 / sqrt(tmp)
         do k = 1,n
            v(k,i) = v(k,i) * xnorm
         enddo
C$ann[DoShared("BLOCK")]
         do j = i+1,n
            tmp = 0.0
            do k = 1,n
               tmp = tmp + v(k,i)*v(k,j)
            enddo
            do k = 1,n
               v(k,j) = v(k,j) - tmp*v(k,i)
            enddo
         enddo
      enddo
```

This is a parallel version of the Modified Gram-Schmidt algorithm. It is made up of two nested loops. The outer loop normalizes each vector stored in the matrix *v*. When a vector is normalized, the remaining vectors in the matrix are then corrected by executing the inner loop. These corrections can be done in parallel. By adding, two Fortran-S directive, the code generator is able to generate a SPMD code that will be executed in every processor. The first directive (C$ann[Shared(v)]) specifies that matrix *v* has to be stored in the shared virtual memory, since it will be updated within a parallel loop. Other variables are replicated in the local memory of each processor. Every processor executes the outer loop as well as all assignments that modify a local variable. However, for each outer loop iteration, only one processor updates the shared variable *v(k,i)*). (In the previous example, every processor will write into replicated variables *tmp*

and *xnorm*.) The second directive (C$ann[DoShared("BLOCK")]) indicate that the following loop is a parallel loop. Each processor is in charge of executing a chunk of the iteration space. A detailed description of Fortran-S can be found in [11].

3.1 KOAN Runtime

The KOAN SVM is embedded in the operating system of the iPSC/2. It allows the use of fast and low-level communication primitives as well as a Memory Management Unit (MMU). The KOAN SVM implements the fixed distributed manager algorithm as described in [39] with an invalidation protocol for keeping the shared memory coherent at all times. A detailed description of the KOAN SVM can be found in [34]. Let us now summarize some of the functionalities of the KOAN SVM runtime.

KOAN SVM provides the user with several memory management protocols for efficiently handling special memory access patterns. One of these is when several processors have to write into different locations of the same page. This pattern involves many messages since the page has to move from processor to processor (as with the *ping-pong effect* or *false sharing*). At a cost of adding some new subroutine calls in the parallel code, KOAN can let processors concurrently modify their own copy of a page. Another drawback of shared virtual memory on DMPCs is its inability to run efficiently parallel algorithms that contain a producer/consumers scheme: a page is modified by a processor and then accessed by the other processors. KOAN SVM can efficiently manage this memory access pattern by using the broadcasting facility of the underlying topology of DMPCs (hypercube, 2D-mesh, etc.). All pages that have been modified by the processor in charge of running the producer phase are broadcast to all other processors that will run the consumer phase in parallel. KOAN SVM provides barrier synchronization as well as subroutines to manage critical sections. These features are implemented by using messages instead of shared variables. KOAN is compatible with the NX/2 operating system, i.e. primitives provided by the system can be used simultaneously with KOAN.

We have performed measurements in order to determine the costs of various basic operations for both read and write page faults (the size of a page is 4 *Kbytes*) of the KOAN shared virtual memory. For each type of page fault (read or write), we have tested the best and worst possible situation on different numbers of processors. For a 32-processor configuration, the time required to resolve a read page fault is in the range of 3.412 ms to 3.955 ms. For a write page fault, timing results are in the range of 3.447 ms to 10.110 ms depending on the number of copies that have to be invalidated. These results can be compared with the communication times of the iPSC/2: the latency is roughly 0.3 ms and sending a 4 *Kbytes* message (a page) costs between 2.17 ms and 2.27 ms depending on the number of routing.

3.2 Fortran-S Code Generator

Fortran-S[†] relies on parallel loops to achieve parallelism. Parallel execution is achieved using the SPMD execution model (Single Program Multiple Data). At the beginning of the program execution, a thread is created on each processor and each processor starts to execute the program. One of the main functions of the Fortran-S compiler is to make the SPMD execution to look like a

[†]The prototype compiler has been implemented using the Sigma System [22]

# proc.	With strong coherence					
	100 x 100			200 x 200		
	Times (ms)	Speedup	Eff.	Times (ms)	Speedup	Eff.
1	3112	-	-	12933	-	-
2	1927	1.61	80.75	7323	1.77	88.30
4	1280	2.43	60.78	3975	3.25	81.34
8	1322	2.35	29.43	2284	5.66	70.78
16	3882	0.80	5.01	1446	8.94	55.90
32	5339	0.58	1.82	1928	6.71	20.96
	With weak coherence					
1	3112	-	-	12933	-	-
2	1972	1.58	78.90	7323	1.77	88.30
4	1311	2.37	59.34	4016	3.22	80.51
8	923	3.37	42.15	2305	5.61	70.14
16	921	3.38	21.12	1567	8.25	51.58
32	1151	2.70	8.45	1244	10.40	32.49

Table 1: Performance results for the Jacobi loops.

single threaded execution, by appropriate insertion of synchronization and the correct updating of shared variables. The programming model uses directives to specify shared variables and parallel loops. A shared variable is accessible in read or write from all the processors. A non shared variable is duplicated on all the processors. Since every processor executes the sequential code sections, non-shared variables have always the same value. The iteration space of a parallel loop is distributed over the processor. Each processor only executes a subset of the iteration space. Fortran-S provides several directives to generate efficient parallel code [11].

3.3 Performance

In this section we present the first results obtained using Fortran-S on an Intel iPSC/2 with 32 nodes. The goal of these experiments was to port sequential Fortran 77 programs to Fortran-S and to measure the performance obtained. We did not intended, in those early performance measurements, to modify extensively the applications. Rather, we intended to measure performance of Fortran-S in a straight forward translation from Fortran 77. Very few modifications have been done to the original program. The primary modification was to expose parallel loops in the programs. However no modification of the data structure used in the program was made. Also they were no major modification to the algorithms, so the scalability of some application is not limited by Fortran-S but by the algorithm used in the application. The problem of false-sharing that appears in many applications was solved using a weak coherence protocol.

The first code used is taken from a Jacobi iteration. Table 1 gives the speedups and efficiencies for different problem sizes when using either a strong or a weak cache coherence protocol. For a matrix size set to 100×100, we got a "speed-down" when the number of processors is greater than 16. False sharing could be avoided by using weak coherence protocol. For the same problem size, this cache coherence protocol improves the speedups a little, but the speed-up remains flat. For a larger problem size (200×200) we did not observe such phenomena. However when the number of processors is set to 32, the efficiency is bad (20.71%). The weak cache coherence protocol increases the efficiency to 32.49%. This behavior is observed only for small matrices. For large matrices the

# proc.	With strong coherence					
	100 x 100			200 x 200		
	Times (ms)	Speedup	Eff.	Times (ms)	Speedup	Eff.
1	15694	-	-	127657	-	-
2	7920	1.98	99.08	64037	1.99	99.67
4	4056	3.87	96.73	32292	3.95	98.83
8	2206	7.11	88.93	16522	7.73	96.58
16	3393	4.63	28.91	8982	14.21	88.83
32	4379	3.58	11.20	5196	24.57	76.78
	With weak coherence					
1	15694	-	-	127657	-	-
2	7923	1.98	99.04	64036	1.99	99.68
4	4048	3.88	96.92	32276	3.96	98.88
8	2202	7.13	89.09	16521	7.73	96.59
16	1287	12.19	76.21	8972	14.23	88.93
32	884	17.75	55.48	5206	24.52	76.63

Table 2: Performance results for the matrix multiply.

efficiency is close to the maximum.

The second parallel algorithm we studied is the matrix multiply. Table 2 gives timing results for small matrices (100×100 and 200×200). For larger matrix size, speedups are near from the maximum. This can be seen in this table; for a 32 nodes configuration, speedups increase from 3.58 to 24.57 when the number of matrix elements quadruples. However, for small matrices, the results can be improved by using the weak cache coherence protocol. Indeed, the poor performance is always due to the same effect: "false-sharing". The same table provides timing results when the parallel loop is executing with weak coherence. For the small matrix, the gain in performances is impressive. When the number of processors is set to 32, speedup augments from 3.58 to 17.75.

# proc.	200×200								
	Strong coherence			Weak coherence			Weak+Broadcast		
	Times (s)	Speedup	Eff.	Times (s)	Speedup	Eff.	Times (s)	Speedup	Eff.
1	125.99	-	-	125.99	-	-	125.99	-	-
2	79.34	1.59	79.40	66.34	1.90	66.34	66.69	1.89	94.46
4	64.20	1.96	49.06	37.07	3.40	84.97	37.09	3.40	84.92
8	61.59	2.05	25.57	23.99	5.25	65.65	23.04	5.47	68.35
16	65.49	1.92	12.02	20.61	6.11	38.21	16.85	7.48	46.73
32	78.79	1.60	5.00	23.62	5.33	16.67	14.88	8.47	26.46
	500×500								
1	1986.81	-	-	1986.81	-	-	1986.81	-	-
2	1029.11	1.93	96.53	1007.51	1.97	98.60	1013.20	1.96	98.05
4	562.52	3.53	88.30	517.57	3.84	95.97	522.38	3.80	95.08
8	339.23	5.86	73.21	276.17	7.19	89.93	278.72	7.13	89.10
16	233.10	8.52	53.27	163.98	12.12	75.73	158.97	12.50	78.11
32	205.75	9.66	30.18	124.71	15.93	49.79	101.62	19.55	61.10

Table 3: Performance results for the MGS algorithm.

The last experiment involved the Modified Gram-Schmidt algorithm described above. This algorithm consists of two nested loops. We added some directives in order to improve the efficiency

of the parallel MGS algorithm. The vector, which is modified in the sequential section, is broadcast to every processor, since it will be accessed within the parallel loop. A weak cache coherence protocol is also associated with the inner loop to avoid false sharing. A detailed study of this algorithm can be found in [52, 53]. Table 3 summarizes the results we obtained with different strategies.

Several other parallel algorithms and applications have been ported to KOAN. Their performance results are presented in [54, 10].

4 High Performance Fortran.

Recently an international group of researchers from academia, industry and government labs formed the High Performance Fortran Forum aimed at providing an intermediate approach in which the user and the compiler share responsibility for exploiting parallelism. The main goal of the group has been to design a high-level set of standard extensions to Fortran called, High Performance Fortran (HPF), intended to exploit a wide variety of parallel architectures [28, 40].

The HPF extensions allow the user to carefully control the distribution of data across the memories of the target machine. However, the computation code is written using a global name space with no explicit message passing statements. It is then the compiler's responsibility to analyze the distribution annotations and generate parallel code inserting communication statements where required by the computation. Thus, using this approach the programmer can focus on high-level algorithmic and performance critical issues such as load balance while allowing the compiler system to deal with the complex low-level machine specific details.

Earlier efforts

The HPF effort is based on research done by several groups, some of which are described below. The language IVTRAN [47], for the SIMD machine ILLIAC IV, was one of the first languages to allow users to control the data layout. The user could indicate the array dimensions to be spread across the processors and those which were to be local in a processor. Combinations resulting in physically skewed data were also allowed.

In the context of MIMD machines, Kali (and its predecessor BLAZE) [45, 46] was the first language to introduce user-specified distribution directives. The language allows the dimensions of an array to be mapped onto an explicitly declared processor array using simple regular distributions such as *block, cyclic* and *block-cyclic* and more complex distributions such as *irregular* in which the address of each element is explicitly specified. Simple forms of user-defined distribution are also permitted. Kali also introduced the idea of dynamic distributions which allow the user to change the distribution of an array at runtime. The parallel computation is specified using *forall* loops within a global name space. The language also introduced the concept of an *on clause* which allows the users to control the distribution of loop iterations across the processors.

The Fortran D project [19] follows a slightly different approach to specifying distributions. The distribution of data is specified by first aligning data arrays to virtual arrays knows as decompositions. The decompositions are then distributed across an implicit set of processors using relative weights for the different dimensions. The language allows an extensive set of alignments along

with simple regular and irregular distributions. All mapping statements are considered executable statements, thus blurring the distinction between static and dynamic distributions.

Vienna Fortran [14, 68] is the first language to provide a complete specification of distribution constructs in the context of Fortran. Based largely on the Kali model, Vienna Fortran allows arrays to be aligned to other arrays and which are then distributed across an explicit processor array. In addition to the simple regular and irregular distributions, Vienna Fortran defines a generalized block distribution which allows unequal sized contiguous segments of the data to be mapped the processors. Users can define their own distribution and alignment functions which can then be used to provide a precise mapping of data to the underlying processors. The language maintains a clear distinction between distributions that remain static during the execution of a procedure and those which can change dynamically, allowing compilers to optimize code for the different the two situations. It defines multiple methods of passing distributed data across procedure boundaries including inheriting the distribution of the actual arguments. Distribution inquiry functions facilitate the writing of library functions which are optimal for multiple incoming distributions.

High Performance Fortran effort has been based on the above and other related projects [8, 27, 38, 48, 58, 59, 60]. In the next few sub-sections we provide, short introduction to HPF concentrating on the features which are critical to parallel performance.

4.1 HPF Overview

High Performance Fortran[‡] is a set of extensions for Fortran 90 designed to allow specification of data parallel algorithms. The programmer annotates the program with distribution directives to specify the desired layout of data. The underlying programming model provides a global name space and a single thread of control. Explicitly parallel constructs allow the expression of fairly controlled forms of parallelism, in particular data parallelism. Thus, the code is specified in high level portable manner with no explicit tasking or communication statements. The goal is to allow architecture specific compilers to generate efficient code for a wide variety of architectures including SIMD, MIMD shared and distributed memory machines.

Fortran 90 was used a base for HPF extensions for two reasons. First, a large percentage of scientific codes are still written in Fortran (Fortran 77 that is) providing programmers using HPF with a familiar base. Second, the array operations as defined for Fortran 90 make it eminently suitable for data parallel algorithms.

Most of the HPF extensions are in the form of directives or structured comments which assert facts about the program or suggest implementation strategies such as data layout. Since these are directives they do not change the semantics of the program but may have a profound effect on the efficiency of the generated code. The syntax used for these directives such that if HPF extensions are at some later date accepted as part of the language only the prefix, **!HPF$**, needs to be removed to retain a correct HPF program. HPF also introduces some new language syntax in the form of data parallel execution statements and a few new intrinsics.

[‡]This chapter is partially based on the High Performance Fortran Language Specification draft document [28] which has been jointly written by several of the participants of the High Performance Fortran Forum. Also, the specification (as described here) are still under review and may change when the final document is released.

Features of High Performance Fortran

In this subsection we provide a brief overview of the new features defined by HPF. In the next few subsections we will provide a more detailed view of some of these features.

- *Data mapping directives:* HPF provides an extensive set of directives to specify the distribution and alignment of arrays.
- *Data parallel execution features:* The **FORALL** statement and construct and the **INDEPENDENT** directive can be used to specify data parallel code. The concept of *pure* procedures callable from parallel constructs has also been defined.
- *New intrinsic and library functions:* HPF provides a set of new intrinsic functions including system functions to inquire about the underlying hardware, mapping inquiry functions to inquire about the distribution of the data structures and a few computational intrinsic functions. A set of new library routines have also been defined so as to provide a standard interface for highly useful parallel operations such as reduction functions, combining scatter functions, prefix and suffix functions, and sorting functions.
- *Extrinsic procedures:* HPF is well suited for data parallel programming. However, in order to accommodate other programming paradigms, HPF provides *extrinsic* procedures. These define an explicit interface and allow codes expressed using a different paradigm, such as an explicit message passing routine, to be called from an HPF program.
- *Sequence and storage association:* The Fortran concepts of sequence and storage association[§] assume an underlying linearly addressable memory. Such assumptions create a problem in architectures which have a fragmented address space and are not compatible with the data distribution features of HPF. Thus, HPF places restrictions on the use of storage and sequence association for distributed arrays. For example, arrays that have been distributed can not be passed as actual arguments associated with dummy arguments which have a different rank or shape. Similarly, arrays that have been storage associated with other arrays can be distributed only in special situations. The reader is referred to the HPF Language specification document [28] for full details of these restrictions and other HPF features.

4.2 Data Mapping Directives

A major part of the HPF extensions are aimed at specifying the alignment and distribution of the data elements. The underlying intuition for such mapping of data is as follows. If the computations on different elements of a data structure are independent, then distributing the data structure will allow the computation to be executed in parallel. Similarly, if elements of two data structures are used in the same computation, then they should be aligned so that they reside in the same processor memory. Obviously, the two factors may be in conflict across computations, giving rise to situations where data needed in a computation resides on some other processor. This data dependence is then satisfied by communicating the data from one processor to another. Thus, the

[§]Informally, sequence association refers to the Fortran assumption that the elements of an array are in particular order (column-major) and hence allows redimensioning of arrays across procedure boundaries. Storage association allows COMMON and EQUIVALENCE statements to constrain and align data items relative to each other.

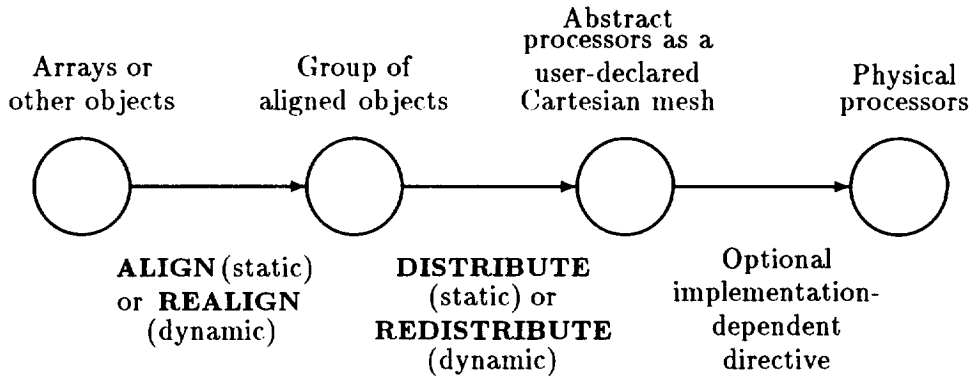


Figure 2: HPF data distribution model

main of goal of mapping data onto processor memories is to increase parallelism while minimizing communication such that the workload across the processors is balanced.

HPF uses a two level mapping of data objects to abstract processors as shown in Figure 2. First, data objects are aligned to other objects and then groups of objects are distributed on a rectilinear arrangement of abstract processors.

Each array is created with some mapping of its elements to abstract processors either on entry to a program unit or at the time of allocation for allocatable arrays. This mapping may be specified by the user through the **ALIGN** and **DISTRIBUTE** directives or in the case where complete specifications are not provided may be chosen by the compiler.

Processors Directive

The **PROCESSORS** directive can be used to declare one or more rectilinear arrangements of processors in the specification part of a program unit. If two processor arrangements have the same shape, then corresponding elements of the two arrangements are mapped onto the same physical processor thus ensuring that objects mapped to these abstract processors will reside on the same physical processor.

The intrinsics **NUMBER_OF_PROCESSORS** and **PROCESSOR_SHAPE** can be used to determine the actual number of physical processors being used to execute the program. This information can then be used in declaring the abstract processor arrangement.

```
!HPF$ PROCESSORS  P(N)
!HPF$ PROCESSORS  Q( NUMBER_OF_PROCESSORS())
!HPF$ PROCESSORS  R(8,NUMBER_OF_PROCESSORS()/8)
!HPF$ PROCESSORS  SCALARPROC
```

Here, P is a processor arrangement of size N, the size of Q (and the shape of R) is dependent upon the number of physical processors executing the program while *SCALARPROC* is conceptually treated as a scalar processor.

A compiler must accept any processor declaration which is either scalar or whose total number of elements match the number of physical processors. The mapping of the abstract processors to physical processors is compiler-dependent. It is expected that implementors may provide architecture-specific directives to allow users to control this mapping.

Distribution Directives

The **DISTRIBUTE** directive can be used to specify the distribution of the dimensions of an array to dimensions of an abstract processor arrangement. The different types of distributions allowed by HPF are: *BLOCK(expr), CYCLIC(expr)* and *.

```
      PARAMETER (N = NUMBER_OF_PROCESSORS())

!HPF$ PROCESSORS  Q(NUMBER_OF_PROCESSORS())
!HPF$ PROCESSORS  R(8,NUMBER_OF_PROCESSORS()/8)

      REAL A(100), B(200), C(100,200), D(100, 200)

!HPF$ DISTRIBUTE  A(BLOCK) ONTO Q
!HPF$ DISTRIBUTE  B(CYCLIC(5))
!HPF$ DISTRIBUTE  C(BLOCK, CYCLIC) ONTO R
!HPF$ DISTRIBUTE  D(BLOCK(10), *) ONTO Q
```

In the above examples, A is divided into N contiguous blocks of elements which are then mapped onto successive processors of the arrangement Q. The elements of array B are first divided into blocks of 5, which are then mapped in a wrapped manner across the processors of the arrangement Q. The two dimensions of array C are individually mapped to the two dimensions of the processor arrangement R. The rows of C are blocked while the columns are cyclically mapped. The one-dimensional array D is distributed across the one-dimensional processor arrangement Q such that the second axis is not distributed. That is each row of the array is mapped as a single object. To determine the distribution of the dimension, the rows are first blocked into groups of 10 and these groups are then mapped to successive processors of Q. In this case, N must be at least 10 to accommodate the rows of D. Note, that in the case of array B, the compiler chooses the abstract processor arrangement for the distribution.

The **REDISTRIBUTE** directive is syntactically similar to the **DISTRIBUTE** directive but may appear only in the execution part of a program unit. It is used for dynamically changing the distribution of an array and may only be used for arrays that have been declared as **DYNAMIC**. The only difference between **DISTRIBUTE** and **REDISTRIBUTE** directives is that the former can use only specification expressions while the latter can use any expression including values computed at runtime.

```
      REAL A(100)
!HPF$ DISTRIBUTE (BLOCK), DYNAMIC :: A
      k = ...
!HPF$ REDISTRIBUTE A(CYCLIC(k))
```

Here, A starts with a block distribution and is dynamically remapped to a cyclic distribution whose block size is computed at runtime.

When an array is redistributed, arrays that are *ultimately aligned* to it (see next subsection) are also remapped to maintain the alignment relationship.

Alignment Directives

The **ALIGN** directive is used to indirectly specify the mapping of an array (the alignee) by specifying its relative position with respect to another object (the align-target) which is ultimately distributed. HPF provides a variety of alignments including identity alignment, offsets, axis collapse, axis transposition, and replication using dummy arguments which range over the entire index range of the alignee. Only linear expressions are allowed in the specification of the align-target with the restriction that a align dummy can appear only in one expression in an **ALIGN** directive. The alignment function must be such that alignee is not allowed to "wrap around" or "extend past the edges" of the align-target.

```
!HPF$ ALIGN A(:,:) WITH B(:,:)      ! identity alignment
!HPF$ ALIGN C(I)   WITH D(I-5)      ! offset
!HPF$ ALIGN E(I,*) WITH F(I)        ! collapse
!HPF$ ALIGN G(I)   WITH H(I,*)      ! replication
!HPF$ ALIGN R(I,J) WITH S(J,I)      ! transposition
```

If A is aligned to B which is in turn aligned with C then A is considered to be *immediately aligned* to B but *ultimately aligned* to C. Note, that intermediate alignments are useful only to provide the "ultimate" alignment since only the root of the alignment tree can be distributed.

The **REALIGN** directive is syntactically similar to the **ALIGN** directive but may appear only in the execution part of a program unit. It is used for dynamically changing the alignment of an array and may only be used for arrays that have been declared as **DYNAMIC**. As in the case of **REDISTRIBUTE**, the **REALIGN** directive can use computed values in its expression. Note, that only an object which is not the root of an alignment tree can be explicitly realigned and that such a realignment does not affect the mapping of any other array.

Template Directive

In certain codes, we may want to align arrays to an index space which is larger than any of the data arrays declared in the program. HPF introduces the concept of *template* as an abstract index space. Declaration of templates uses the keyword **TEMPLATE** and a syntax similar to that of regular data arrays. The distinction is that templates do not take any storage.

Consider the situation where two arrays of size $N \times (N+1)$ and $(N+1) \times N$ have to be aligned such that bottom right corner elements are mapped to the same processor. This can be done as follows:

```
!HPF$ TEMPLATE T(N+1,N+1)

!HPF$ REAL A(N,N+1), B(N+1,N)
!HPF$ ALIGN A(I,J) WITH T(I+1,J)
!HPF$ ALIGN B(I,J) WITH T(I,J+1)
!HPF$ DISTRIBUTE  T(BLOCK,BLOCK)
```

As seen above, templates can be used as align-targets and may be distributed using a **DISTRIBUTE** (or **REDISTRIBUTE**) directives but may not be an alignee.

Procedure Boundaries

HPF allows distributed arrays to be passed as actual arguments to procedures. As noted before, HPF places restrictions on sequence association, therefore the rank and shape of the actual arguments must match with those of the corresponding dummy arguments. HPF provides a wide variety of options to specify the distribution of the dummy argument. The user can specify that the distribution of the actual argument be inherited by the dummy argument. In other cases, the user can provide a specific mapping for the dummy and actual argument may need to remapped to satisfy this mapping. If the actual is remapped on entry, then the original mapping is restored on exit from the procedure. The user can also demand that the actual argument be already mapped as specified for the dummy argument. In this case, it is incumbent upon the callee to explicitly remap before the call to the procedure. In the presence of interface blocks such a remap may be implicitly provided by the compiler.

HPF also provides a **INHERIT** directive which specifies that the template of the actual argument be copied and used as the template for the dummy argument. This makes a difference when only a subsection of an array is passed as an actual argument. Without the **INHERIT** directive, the template of the dummy argument is implicitly assumed to be the same shape as the dummy and the dummy is aligned to the template using the identity mapping.

4.3 Data Parallel Constructs

Fortran 90 has syntax to express data parallel operations on full arrays. For example, the statement $A = B + C$ indicates that the two arrays B and C should be added element by element (in any order) to produce the array A. The two main reasons for introducing these features is the conciseness of the expressions (note the absence of explicit loops) and the possibility of exploiting the undefined order of elemental operations for vector and parallel machines. HPF extends Fortran 90 with several new features to explicitly specify data parallelism. The **FORALL** statement and construct generalize the Fortran 90 array operations to allow not only more complicated array sections but also the calling of *pure* procedures on the elements of arrays. The **INDEPENDENT** directive can be used to specify parallel iterations.

Forall Statement

The **FORALL** statement extends the Fortran 90 array operations by making the index used to range over the elements explicit. Thus, this statement can be used to make an array assignment to

array elements or sections of arrays, possibly masked with a scalar logical expression. The general form the **FORALL** statement is as follows:

FORALL (*triplet, ...* [, *scalar-mask*])
assignment

where, a *triplet* has the form:

subscript = *lower*: *upper* [: *stride*]

Here, the **FORALL** header may have multiple triplets and *assignment* is a arithmetic or pointer assignment. First the lower bound, upper bound and the optional stride of each triplet are evaluated (in any order). The cartesian product of the result provides the valid set of subscript values over which the mask is then evaluated. This gives rise to the active combinations. The right hand side of the assignment is then evaluated for all the active combinations before any assignment to corresponding elements on the left hand side.

```
FORALL (I=1,N, J=2,N)
   A(I,J) = A(I,J-1)*B(I)
```

In the above example, the new values of the array A are determined by the old values of A in the columns on the right and the array B.

Forall Construct

The **FORALL** construct is a generalization of the **FORALL** statement allowing multiple statements to be associated with the same forall header. The only kind of statements allowed are assignment, the **WHERE** statement and another **FORALL** statement or construct.

FORALL (*triplet, ...* [, *scalar-mask*])
statement
...
END FORALL

Here, the header is evaluated as before and the execution of one statement is completed for all active combination before proceeding to the next statement. Thus, conceptually in a **FORALL** construct, there is a synchronization before the assignment to the left hand side and between any two statements. Obviously, some of these synchronization may not be needed and can be optimized away.

Pure procedures

HPF has introduced a new attribute for procedures called **PURE** which allows users to declare that the given procedure has no side effects. That is the only effects of the procedure are either the value returned by the function or possible changes in the values of **INTENT(OUT)** or **INTENT(INOUT)** arguments. HPF defines a set of syntactic constraints that must be followed

in order for a procedure to be pure. This allows the compiler to easily check the validity of the declaration. Note, that a procedure can only call other pure procedures to remain pure.

Only pure functions can be called from a **FORALL** statement or construct. Since pure functions have no side-effects other than the value returned, the function can be called for the active set of index combinations in any order.

Independent Directive

The **INDEPENDENT** directive can be used with a **DO** loop or a **FORALL** statement or construct to indicate that there are no cross-iteration data dependences. Thus, for a **DO** loop the directive asserts that the iterations of the loop can be executed in any order without changing the final result. Similarly when used with a **FORALL** construct or statement, the directive asserts that there is no synchronization required between the executions of the different values of the active combination set.

With a **DO** loop, the **INDEPENDENT** directive can be augmented with a list of variables which can be treated as private variables for the purposes of the iterations.

```
!HPF$ INDEPENDENT, NEW(X)
      DO I = 1,N
         X = B(I)
          ...
         A(f(I)) = X
      END DO I = 1,N
```

Here, the **INDEPENDENT** directive is asserting that the function $f(I)$ returns a permutation of the index set, i.e., no two iterations are going to assign to the same element of A. Similarly, the *new clause* asserts that the loop carried dependence due to the variable X is spurious and the compiler can execute the loops by (conceptually) allocating a *new* X variable for each iteration.

4.4 Examples of HPF Codes

In this section we provide two code fragments using some of the HPF features described above. The first is the Jacobi iterative algorithm and the second is the Modified Gram-Schmidt algorithm discussed earlier.

The HPF version of the Jacobi iterative procedure which may be used to approximate the solution of a partial differential equation discretized on a grid, is given below.

```
!HPF$ processors  p(number_of_processors())
       real u(1:n,1:n), f(1:n,1:n)
!HPF$ align u :: f
!HPF$ distribute u (*, block)
      forall (i=2:n-1, j = 2:n-1)
           u(i,j) = 0.25 * (f(i,j) + u(i-1, j) + u(i+1, j) +
                                u(i, j-1) + u(i, j+1)
      end forall
```

At each step, it updates the current approximation at a grid point, represented by the array u, by computing a weighted average of the values at the neighboring grid points and the value of the right hand side function represented by the array f.

The array f is aligned with the array u using the identity alignment. The columns of u (and thus those of f indirectly) are then distributed across the processors executing the program. The computation is expressed using a **FORALL** statement, where all the right hand sides are evaluated using the old values of u before assignment to the left hand side.

To reiterate, the computation is specified using a global index space and does not contain any explicit data motion constructs. Given that the underlying arrays are distributed by columns, the edge columns will have to be communicated to neighboring processors. It is the compiler's responsibility to analyze the code and generate parallel code with appropriate communication statements inserted to satisfy the data requirements.

The HPF version of the Modified Gram-Schmidt algorithm is given below.[¶].

```
      real v(n,n)
!HPF$ distribute v (*, block)
      do i = 1,n
       tmp = 0.0
       do k = 1,n
         tmp = tmp + v(k,i)*v(k,i)
       enddo
       xnorm = 1.0 / sqrt(tmp)
       do k = 1,n
         v(k,i) = v(k,i) * xnorm
       enddo
!HPF$ indepedent, new(tmp)
       do j = i+1,n
         tmp = 0.0
         do k = 1,n
           tmp = tmp + v(k,i)*v(k,j)
         enddo
         do k = 1,n
           v(k,j) = v(k,j) - tmp*v(k,i)
         enddo
       enddo
      enddo
```

The first directive declares that the columns of the array v are to be distributed by block across the memories of the underlying processor set. The outer loop is sequential and is thus executed by all processors. Given the column distribution, in the ith iteration of the outer loop, the first two k loops would be executed by the processor owning the ith column.

[¶]A Fortran 90 version of the code fragment, not shown here, would have used array constructs for the k loops. This would make the parallelism in the inner loops explicit.

The second directive declares the j loop to be *independent* and *tmp* to be a *new* variable. Thus the iterations of the j loop can be executed in parallel, i.e., each processor updates the columns that it owns in parallel. Since the ith column is used for this update, it will have to be broadcast to all processors.

The distribution of the columns by contiguous blocks implies that processors will become idle as the computation progresses. A *cyclic* distribution of the columns would eliminate this problem. This can be achieved by replacing the distribution directive with the following:

!HPF$ distribute v (*, cyclic)

This declares the columns to distributed cyclically across the processors, and thus forces the inner j loop to be strip-mined in a cyclic rather than in a block fashion. Thus, all processors are busy until the tail end of the computation.

The above distributions only exploit parallelism in one dimension, whereas the inner k loops can also run in parallel. This can be achieved by distributing both the dimensions of v as follows:

!HPF$ distribute v (block, cyclic)

Here, the processors are presumed to be arranged in a two-dimensional mesh and the array is distributed such that the elements of a column of the array are distributed by block across a column of processors whereas the columns as a whole are distributed cyclically. Thus, the first k loop becomes a parallel reduction of the ith column across the set of processors owning the ith column. Similarly, the second k loop can be turned into a **FORALL** statement which is executed in parallel by the column of processors which owns the ith column. The second set of k loops, inside the j loop, can be similarly parallelized.

Overall, it is clear, that using the approach advocated by HPF allows the user to focus on the performance critical issues at a very high level. Thus, it is easy for the user to experiment with a different distribution, by just changing the distribute directives. The new code is then recompiled before running on the target machine. In contrast, the effort required to change the program if it was written using low-level communication primitives would be much more.

5 Object Parallelism with pC++

pC++ is an experimental extension to C++ designed to allow programmers to build distributed data structures with parallel execution semantics. These data structures are organized as "concurrent aggregate" collection classes which can be aligned and distributed over the memory hierarchy of a parallel machine is a manner modeled on the High Performance Fortran Forum (HPF) directives for Fortran 90. The first version of the compiler is a preprocessor which generates Single Program Multiple Data (SPMD) C++ code which runs on the Thinking Machines CM-5, the Intel Paragon, the BBN TC2000 and the Sequent series of machines. As HPF becomes available on these systems future versions of the compiler will allow object level linking between pC++ distributed collections and HPF distributed arrays.

The basic concept of pC++ is the notion of a distributed collection, which is a type of concurrent aggregate "container class" [15, 35]. More specifically, a *collection* is a structured set of objects

distributed across the processing elements of the computer. A runtime system uses the memory hierarchy and processor interconnect topology of the target machine to guide the distribution of collection elements. A collection can be an *Array*, a *Grid*, a *Tree*, or any other partitionable data structure.

Collections have the following components:

- A collection class describing the basic topology of the set.
- A size or shape for each instance of the collection class. For example, the dimensions of an array or the height of a tree.
- A base type for collection elements. This can be any C++ type or class. For example, one can define an Array of *Floats*, or a Grid of *FiniteElements*, or Matrix of *Complex*, or a Tree of *X*s, where *X* is the class of each node in the tree.
- A *Distribution* object. The distribution describes an abstract coordinate system that will be distributed over the available memory modules of the target by the run-time system.
- A function object called the *Alignment*. This function maps collection elements to the abstract coordinate system of the *Distribution* object.

The pC++ language has a library of standard collection classes that may be used (or subclassed) by the programmer [36, 49, 17, 20]. This includes collection classes such as *DistributedArray*, *DistributedMatrix*, *DistributedVector*, and *DistributedGrid*. To illustrate the points above, consider the problem of creating a distributed 5 by 5 matrix of floating point numbers. We begin by building a *Distribution*. A distribution is defined by its number of dimensions, the size in each dimension and how the elements are mapped to the processors. In HPF [28] this mapping is called a distribution. Current distributions include BLOCK, CYCLIC and WHOLE, but more general forms will be added later. Let us assume that the distribution is distributed over the processor's memories by mapping *Whole* rows of the distribution to individual processors using a *Cyclic* pattern where the i^{th} row is mapped to processor memory i mod P, on a P processor machine.

pC++ uses a special implementation dependent library class called *Processors*. In the current implementation, it represents the set of all processors available to the program at run time. To build a distribution of some size, say 7 by 7, one would write

```
Processor P;
Distribution myDist(7, 7, &P, Cyclic, Whole);
```

Next, we create an alignment object called *myAlign* that defines a domain and function for mapping the matrix to the distribution. The matrix `A` can be defined using the library collection class *DistributedMatrix* with a base type of *Float*.

```
Align myAlign(5, 5, "[ALIGN( domain[i][j], myDist[i][j])]" );
DistributedMatrix<Float>  A(myDist, myAlign);
```

The collection constructor uses the alignment object, *myAlign*, to define the size and dimension of the collection. The mapping function is described by a text string corresponding to the HPF

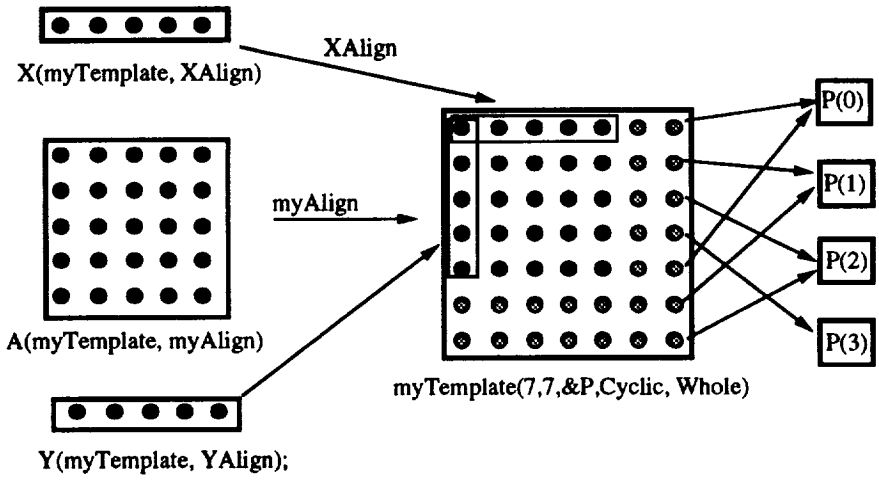


Figure 3: Alignment and Distribution

alignment directive. It defines a mapping from a domain structure to a distribution structure using dummy index variables.

The intent of this two stage mapping, as it was originally designed for HPF, is to allow the distribution to be a frame of reference so that different arrays could be aligned with each other in a manner that promotes memory locality. For example, suppose we wish to perform a matrix vector multiply. Since the *DistributedMatrix* and *DistributedVector* library classes provide many common functions through C++ function overloading, a matrix vector multiply is simply written as

```
Y = A*X;
```

where X and Y are distributed arrays. While the semantic meaning and computed result of the expression is independent of alignment and distribution, performance is best if the alignment of the operands matches the library function for matrix vector multiply. In this case, the algorithm broadcasts the vector operand along the columns of the array and then performs a reduction along rows. Aligning X along with the first row of the matrix A, and Y with the first column yields the best performance. The vectors are declared by

```
Align XAlign(5, "[ALIGN( X[i], myDist[0][i])]");
Align YAlign(5, "[ALIGN( Y[i], myDist[i][0])]");
DistributedVector<Float>  X(myDist, XAlign);
DistributedVector<Float>  Y(myDist, YAlign);
```

The two stage mapping process for this example is illustrated in Figure 4.

5.1 Collection Functions and Parallelism

There are two forms of concurrency in pC++. One is based on the concurrent application of a method function, associated with the element class across the entire collection, and the other type is associated with special functions that are invoked as a set of parallel threads one running on each processor. More precisely, a collection is a set of element objects. A local collection is the subset of elements mapped to one processor by the alignment and distribution functions. Each

local collection is realized as a *Processor Object* and there is an associated thread of computation that executes all method functions that modify or access the local elements.

The memory model used by pC++ is not shared. As with HPF Fortran, there is a single main thread of computation and parallel operations are invoked from that thread. Collection elements are distributed over the processor objects which each have a private address space. Global data, which can be accessed and modified by the main thread is visible to the processor objects, but a processor object cannot modify Global data. Each processor object can read and write its local collection of elements, but the only way a processor object or the main thread of execution can access remote collection elements is through special kernel functions which which provide a copy of remote collection elements.

A collection class C is a data type that is parameterized by the class of the element, $C < ElementType >$. Collections have two types of methods: the standard *public*, *private* and *protected* methods of any normal class; and a set of fields and methods that are added to the element class to provide access to the collection structure. This additional family of fields and methods are called *MethodOfElement* fields.

Syntactically, a collection class takes the form:

```
collection CollectionName: ParentCollection {
     public:
     private:
     protected:
       // Field variables declared here are local to each
       // processor object.
       // Methods declared here are executed in parallel by
       // the associated processor object thread.
     MethodOfElement:
       // Field variables declared here are added to each element
       // Methods declared here are added to the element class.
       // These methods are the "data parallel" functions.
     }
```

Data fields defined in the public, private and protected areas are duplicated in each processor object. Methods in these areas are executed by the threads of the processor objects.

5.2 An Example: The Gram-Schmidt Algorithm

To illustrate these ideas we will consider the same Gram-Schmidt algorithm discussed earlier. pC++ programmers work by building collections classes derived from the base library. Because Gram-Schmidt works on column vectors of a matrix, we will cast our matrix as a distributed collection of column vectors. Consequently, we shall assume we have a library of double precision vectors which have all the standard vector-vector and vector-scalar operators,

```
class Vector{
   public:
      Vector(int n);  // a constructor.
      Vector & operator *=(double);  // V  = V * 3.14
      double dotProduct(Vector *);   // the dot product
      Vector & operator -=(Vector);  // V =  V -  W
      Vector operator *(double);     // mult. expression
      ...
```

We will define a collection *MyMatrix* which will be a distributed array of elements of class *Vector*. The matrix object and the Gram-Schmidt operation will be invoked as

```
main(){
   Processor P;
   int n = 100;
   Distribution myDist(n, &P, Cyclic);
   Align myAlign(n, "[ALIGN( domain[i], myDist[i])]" );
   MyMatrix<Vector>  M(myDist, myAlign, n);
   M.gramSchmidt(n);
 }
```

This declares M to be a *MyMatrix* collection of size n of elements of class *Vector*. The extra parameter n on the declaration of M is passed to the element constructor so that each vector element has size n. The function *gramSchmidt()* will be a processor object parallel function of the collection which is defined as

```
collection MyMatrix: DistributedArray{
   public:
     void GramSchmidt(int n);
   MethodOfElement:
     void update(ElementType *);
     virtual ElementType & operator *=(double);
     virtual double dotProduct(ElementType *);
     virtual ElementType & operator -=(ElementType);
     virtual ElementType operator *(double);
   }
```

The element level, data parallel functions in this collection include a method *update* which will be described below, and four virtual functions which are provided by the element class which, in our case, is *Vector*. Because the collection is defined separately from the element, if we wish to assume the element has special properties, these are listed as virtual functions. In the case of the Gram-Schmidt algorithm we need to be able to compute the dot product of vectors, multiply vectors by a scalar and subtract a multiple of one vector from another.

The Gram-Schmidt function is nearly a direct translation of the program in section 3.0.

```
void MyMatrix::gramSchmidt(int n){
    ElementType *v;
    int i;
    double temp;
    for(i = 0; i < n; i++){
          v = this->Get_Element(i);
          temp = v->dotProduct(v);
          v *= 1.0 / sqrt(temp);
          (*this)[i+1 : n-1].update(v);
         }
    }
```

In this program *gramSchmidt(n)* is a collection public function which means that it is invoked on each processor object. The main loop first extract the i^{th} column vector element. The pointer v obtained by the kernel function *Get_Element(i)* references a copy of the i^{th} element if it is not part of the local collection of the invoking processor object. Otherwise, v references the actual element. Notice that each processor thread then duplicates the work of computing the dot product and normalizing its copy of v.

The element function *update(v)* is invoked in "data parallel" mode on each element in the local collection that has indexes in the given subrange. In pC++ this is accomplished with an expression of the form

```
collection . elementMethod()
```

which invokes the element method function "in parallel" on each of the elements of the collection. To invoke the method on a subrange we use a Fortran 90 style triplet

```
collection [ lower : upper : stride ] . elementMethod()
```

The parallel operation *update* is identical to the "DoShared" loop in the Fortran-S program.

```
void MyMatrix::update(ElementType *v){
       double temp;
       temp = this->dotProduct(v);
       *this -= v*temp;
};
```

There are two further observations that should be made about this program. First, the use of *Get_Element()* by each processor object can create a serial section. Each processor object other than the owner of the i^{th} element will request a a copy. A more efficient program would use a coordinated element broadcast, *Element_Broadcast()*, to make sure each processor object would get a copy in the smallest amount of time. Second, and more important, is the choice of data distribution. In our case we have selected a cyclic distribution so that as i increases in the expression $[i+1 : n-1].update(v)$, a majority of processors can participate for as long as possible. A block distribution would decrease the parallelism much faster.

6 Conclusion

In this paper we have examined three different approaches to programming scientific, data-parallel applications.

Fortran-S plus SVM provides the user with a familiar model: Fortran 77 plus annotations to distribute loops over processors. Initial experiments with the KOAN SVM system look very promising, but we need much more experience with large applications on large new systems before we can declare success. In the future we expect that more shared virtual memory systems will be implemented on a variety of massively parallel systems. While the details of each system will vary, the Fortran-S project demonstrates that the compiler technology exists to make this model work.

High Performance Fortran provides a high level approach to data parallel programming for a wide variety of architecture. Initial experience has shown that the directives as currently provided by HPF are adequate for simple scientific codes. However, it is also clear that HPF does not have enough expressive power to specify the distributions required for other types of codes such as multi-block and unstructured computations, adaptive computations and multi-disciplinary applications which require integrating different types of parallel programming paradigms.

Currently there are no existing compilers for HPF; several vendors have promised initial implementations in the near future. However, several research projects have built prototype compilers for HPF-like languages. This includes the Kali compiler [33], the SUPERB project [67] on which the Vienna Fortran compiler is based, the Fortran D compiler [29] and several other efforts [8, 24, 27, 32, 38, 48, 58, 59, 60] that have contributed to the overall goal of compiling global name space programs for distributed memory SIMD and MIMD machines.

pC++ is just one example of a number of efforts to add parallelism to C++. While pC++ has been ported to a wide variety of machines including the TMC CM-5, Intel Paragon, BBN TC2000 and the KSR-1, it does have serious drawbacks. First, it relies on an extension to the C++ language. While not a large departure from C++, the collection plus processor object model requires considerable sophistication on the part of the user to use correctly. Also the common alternative, building class libraries that operate in SPMD parallel execution is very popular and it does not require extensions to the language. In the future, the success of object parallel extension to C++ will depend on providing more functionality than Fortran-S or HPF. The feature that will be important are heterogeneous (polymorphic), dynamic collections and nested data parallelism.

We have not attempted a complete survey of the parallel programming landscape. The three parallel programming language extensions described here represent only a small fraction of the approaches currently being investigated. It is clear that this is an area that will continue to undergo rapid evolution. Different application areas may require different programming paradigms and some multi-disciplinary problems will need a combination of programming styles.

References

[1] Scalable coherence interface. Technical report, IEEE Standard P1596, 1991.

[2] KSR parallel programming. Technical report, Kendall Square Research Corporation, February 1992.

[3] Porterfield A. Compiler management of program locality. *Technical Report, Rice University, Houston, Texas*, January 1988.

[4] Kuck D. Abu-Sufah W. and Lawrie D. On the performance enhancement of paging system through program analysis and transformations. *IEEE Transactions on COmputers*, May 1981.

[5] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19:26–34, August 1986.

[6] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. Research Report RC 13115 (#56866), IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1987.

[7] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4), October 1987.

[8] F. André, J.-L. Pazat, and H. Thomas. PANDORE: A system to manage data distribution. In *International Conference on Supercomputing*, pages 380–388, June 1990.

[9] F. Bodin, C. Eisenbeis, W. Jalby, and D. Windheiser. A quantitative algorithm for data locality optimization. In *Code Generation-Concepts, Tools, Techniques*. Springer Verlag, 1992.

[10] F. Bodin, J. Erhel, and T. Priol. Parallel sparse matrix vector multiplication using a shared virtual memory environment. In *Proceeeding of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, March 1993.

[11] F. Bodin, L. Kervella, and T. Priol. Fortran-s: A fortran interface for shared virtual memory architectures. In *Supercomputing'93*, pages 274–283. IEEE, November 1993.

[12] L.M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. on Computers.*, C-27(12):1112–1118, Dec 1978.

[13] B. Chapman, P. Mehrotra, J. Van Rosendale, and H. Zima. A software architecture for multidisciplinary applications: Integrating task and data parallelism. ICASE Report 94-18, NASA CR No. 194896, Institute for Computer Applications in Science and Engineering, Hampton, VA, 1994.

[14] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[15] A. Chien and W. Dally. Concurrent aggregates (CA). In *Second ACM Sigplan Symposium on Principles & Practice of Parallel Programming*. ACM, 1990.

[16] Granston E. D. and Veidenbaum A. V. Integrated hardware/software solution for effective management of local storage in high-performance systems. *Proceedings of the 1991 International Conference on Parallel Processing*, 2:83–90, 1991.

[17] J. K. Lee D. Gannon. On using object oriented parallel programming to build distributed algebraic abstractions. In *Conpar-Vap*, pages 769–774. Springer Verlag, 1992.

[18] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *International Conference on Parallel Processing*, pages 377–380, 1991.

[19] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Department of Computer Science Rice COMP TR90079, Rice University, March 1991.

[20] D. Gannon. Libraries and tools for object parallel programming. In *Advances in Parallel Computing: CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing, Saint Hilaire du Touvet*, volume 6, pages 231–246. Elsevier Science Publisher, 1993.

[21] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global programming transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988. special issue on languages, compilers, and environments for parallel programming.

[22] Dennis Gannon, Jenq Kuen Lee, Bruce Shei, Sekhar Sarukaiand Srivinas Narayana, Neelakantan Sundaresan, Daya Atapattu, and François Bodin. Sigma ii: A tool kit for building parallelizing compilers and performance analysis systems. *Programming Environments for Parallel Computing, IFIP Transactions A-11*, pages 17–36, 1993.

[23] Gallivan K. Gannon D, Jalby W. Strategies for cache and local memory management by global program transformation. *Proceedings of the International Conference on Supercomputing, Springer Verlag, New York, 1987 and Journal of Parallel and Distributed Computing*, October 1988.

[24] H. M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Bonn, December 1989.

[25] E.D. Granston and H. Wijshoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In *Proceedings of the International Conference on Supercomputing*, page To appear. ACM, 1993.

[26] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. University of Illinois at Urbana-Champaign Technical Report CRHC-90-14, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, October 1990.

[27] P. Hatcher, A. Lapadula, R. Jones, M. Quinn, and J. Anderson. A production quality C* compiler for hypercube machines. In *3rd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 73–82, April 1991.

[28] High Performance FORTRAN Language Specification. Technical report, Rice University, 1993.

[29] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[30] Li J. and Chen M. Generating explicit communication from shared-memory progran references. *Proceedings of Supercomputing*, November, 1990.

[31] C.F. Kesselman K.M. Chandy. CC++: A declarative concurrent object oriented programming notation. In *In Research Directions in Object Oriented Programming.* MIT Press, 1993.

[32] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[33] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execut ion. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[34] Z. Lahjomri and T. Priol. Koan: a shared virtual memory for the ipsc/2 hypercube. In *CONPAR/VAPP92*, September 1992.

[35] J. K. Lee. Object oriented parallel programming paradigms and environments for supercomputers. Technical report, Ph.D. Thesis, DCS, Indiana University, June 1992.

[36] J. K. Lee and D. Gannon. Object oriented parallel programming: Experiments and results. In *Supercomputing 91 (Albuquerque, Nov.)*, pages 273–282. IEEE Computer Society and ACM, 1991.

[37] D. Lenoski, J. Laudon, K. Gharachorloo, A Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. of the 17th Annual Symposium on Computer Architecture*, pages 148–160, New York, May 1990. IEEE.

[38] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Technical Report YALEU/DCS/TR-725, Yale University, New Haven, CT, November 1989.

[39] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986.

[40] D. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1:25–42, February 1993.

[41] O'Boyle M. *Program and Data Transformation for Efficient Execution on Distributed Memory Architectures.* PhD thesis, University of Manchester, 1993.

[42] E. P. Markatos and T.J. Leblanc. Using processo affinity in loop scheduling on shared memory multiprocessors. In *Supercomputing*, pages 104–113, 1992.

[43] J. McGraw, S. Skedzielewski, S. Allan, R. Oldenhoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Language reference manual. Report M-146, Lawrence Livermore National Laboratory, March 1985.

[44] K. S. McKindley. *Automatic and Interactive Parallelization.* PhD thesis, Rice University, 1992.

[45] P. Mehrotra. Programming parallel architectures: The BLAZE family of languages. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 289–299, December 1988.

[46] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 364–384. Pitman/MIT-Press, 1991.

[47] R. E. Millstein. Control structures in ILLIAC IV Fortran. *Communications of the ACM*, 16(10):621–627, October 1973.

[48] *MIMDizer User's Guide, Version 7.02.* Pacific Sierra Research Corporation, Placerville, CA., 1991.

[49] F. Bodin P. Beckman D. Gannon S. Yang S. Kesavan A. Malony B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Supercomputing 93.* IEEE Computer Society, 1993.

[50] D. Mosberger. Memory consistency models. *ACM Operating Systems Review*, February 1993.

[51] D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):763–776, September 1980.

[52] T. Priol and Z. Lahjomri. Experiments with shared virtual memory on a ipsc/2 hypercube. In *International Conference on Parallel Processing*, pages 145–148, August 1992.

[53] T. Priol and Z. Lahjomri. Trade-offs between shared virtual memory and message-passing on an ipsc/2 hypercube. Technical Report 1634, INRIA, 1992.

[54] Because Esprit Project. *Because Test Programs: BBS.2.5.1 (Matrix Assembly)*, 1992.

[55] Gao G. R., Olsen R., Sarkar V., and Thekkath R. Collective loop fusion for array contraction. *5th Workshop on Languages and Compilers for Parallel Computing*, pages 1–31, 92.

[56] J. Ramanujam and P. Sadayappan. Nested loop tiling for distributed memory machines. In *Proceedings of the The Fifth Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[57] M. Raynal and M. Mizuno. How to find his way in the jungle of consistency criteria for distributed objects memories (or how to escape from minos' labyrinth. In *IEEE conference on Future Trends of DCS*, 1993.

[58] A. P. Reeves and C. M. Chase. The Paragon programming paradigm and distributed memory multicomputers. In *Compilers and Runtime Software for Scalable Multiprocessors, J. Saltz and P. Mehrotra Editors*, Amsterdam, The Netherlands, To appear 1991. Elsevier.

[59] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 1–999. ACM SIGPLAN, June 1989.

[60] M. Rosing, R. W. Schnabel, and R. P. Weaver. Expressing complex parallel algorithms in DINO. In *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications*, pages 553–560, 1989.

[61] J. Torrellas, M. S. Lam, and J. L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *International Conference on Parallel Processing*, pages 266–270, August 1990.

[62] Chau-Wen Tseng. *An Optimizing Fortran D compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, 1993.

[63] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings ACM SIGPLAN 91*, June 1991.

[64] Lam M. Wolf M. A data locality optimizing algorithm. *ACM Conference on Programming Language Design and Implementation*, pages 26–28, June 1991.

[65] M. J. Wolfe. More iteration space tiling. In *Supercomputing '89*, November 1989.

[66] J. Wu, J. Saltz, S. Hiranandanu, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, pages 26–30, 1991.

[67] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.

[68] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. Internal Report 21, ICASE, Hampton, VA, March 1992.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1994	Contractor Report

4. TITLE AND SUBTITLE
DIRECTIONS IN PARALLEL PROGRAMMING: HPF, SHARED VIRTUAL MEMORY AND OBJECT PARALLELISM IN pC++

5. FUNDING NUMBERS
C NAS1-19480
WU 505-90-52-01

6. AUTHOR(S)
Francois Bodin, Thierry Priol,
Piyush Mehrotra and Dennis Gannon

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 94-54

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-194943
ICASE Report No. 94-54

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report
To Appear in Proceedings of Summer Inst. on Parallel Comp. Architectures, Lang. and Algorithms, IEEE Press

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Fortran and C++ are the dominant programming languages used in scientific computation. Consequently, extensions to these languages are the most popular for programming massively parallel computers. We discuss two such approaches to parallel Fortran and one approach to C++. The High Performance Fortran Forum has designed HPF with the intent of supporting data parallelism on Fortran 90 applications. HPF works by asking the user to help the compiler distribute and align the data structures with the distributed memory modules in the system. Fortran-S takes a different approach in which the data distribution is managed by the operating system and the user provides annotations to indicate parallel control regions. In the case of C++, we look at pC++ which is based on a concurrent aggregate parallel model.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Data parallel programming, high performance Fortran, Shared virtual memory, object parallelism	39
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298(Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

☆U.S. GOVERNMENT PRINTING OFFICE: 1994 - 528-064/23014